NYU Abu Dhabi **Division of Engineering**

NEW YORK UNIVERSITY ABU DHABI

ENGR-UH 3511

Computer Organization and Architecture

Name: Nishant Aswani

Net ID: nsa325

Assignment Title: Homework 4

ENGR-UH 3511 Homework 4

Nishant Aswani (nsa325@nyu.edu)

September 22, 2019

Question 4.1.1

We are given:

Instruction: `AND Rd, Rs, Rt`
Interpretation: `Reg[RD] = Reg[Rs] AND Reg[Rt]`

There are 7 control signals, each of which depend on the instruction in one way or another:

RegWrite: 1; this is an R type instruction that involves changing the value in a register (result of bitwise AND into Rd).

MemRead: 0; this is enabled for load instructions, where data memory must be accessed. ANDing of register values does not involve any data loading from memory.

ALUMux: select for register data from register file; this is an R-type instruction ANDing two register values, so it requires data from both to be fed into the ALU.

MemWrite: 0; this is enabled for store instructions, where data memory must be updated. Bitwise AND of register values does not involve any updates to data memory.

ALU Operation: code for bitwise and, 0000; depending on the instruction, the ALU is used for an arithmetic operation. Here it is obvious that bitwise AND is required.

RegMux: select for ALU output; because there is no access to data memory, we are looking for the output of the register data addition to store back into the write register.

Branch: 0; This is not a branch instruction. Even if the output of bitwise AND is 0, we do not want that to affect the program counter.

Question 4.1.2

The program counter outputs the address of the instruction at hand. The instruction memory obtains the instruction from this address. The PC +4 Adder is also important to initially get the address for the instruction. The register file obtains the values from the registers and stores the resulting value, while the ALU carries out the AND operation. As always, the control logic block is key in correctly assigning data to ports. If MUXes may be considered as blocks, The ALUMux is important in selecting the register data as a second ALU input, while the RegMux is important in selecting the output of the ALU over the output of data memory.

Question 4.1.3

The data memory block, although it receives input according to Figure 2, produces unused output. This is because the MemRead control is disabled; as a result, it outputs 0 values.

The ALU Zero produces a unused value because the Branch control logic is disabled. As it is not a branch instruction, an output of zero is irrelevant to the program counter.

The adder above the register file produces an output which is unused because there is no branching involved. Hence, the PC increments by 4.

Question 4.3.1

We are given:

"where I-Mem, Add, Mux, ALU, Regs, D-Mem, and Control blocks have latencies of 400 ps, 100 ps, 30 ps, 120 ps, 200 ps, 350 ps, and 100 ps, respectively,and costs of 1000, 30, 10, 100, 200, 2000, and 500, respectively.

This [multiplication feature] will add 300 ps to the latency of the ALU and will add a cost of 600 to the ALU."

The clock cycle time depends on the longest path. The longest path in this implementation is by the load word instruction, leading to a total latency of:

400ps (IM) + 200ps (Regs) + 30ps (ALUMux) + 120ps (ALU) + 350ps (DM) + 30ps (RegMux) + 200ps (Regs) = **1330ps for the original clock cycle time**

With the improvement in the ALU, we can determine the new clock cycle time by adding 300ps. **The updated clock cycle time would be 1630ps.**

Question 4.3.2

We can calculate speedup by taking a ratio of the performance values of the two CPUs, where CPU_1 refers to the updated CPU. Below, T is the clock cycle time and I is the instruction count.

$$Speedup = \frac{CPU_1}{CPU_2} = \frac{T_1 * I_1}{T_2 * I_2} = \frac{1630 * 0.95I}{1330 * I} = 1.16 \tag{1}$$

So, there is a slowdown by implementation of the MUL instruction, since the old implementation is 1.16 times faster.

Question 4.3.3

The original cost is:

1000 (IM) + 30 (Add) + 30 (Add) + 200 (Regs) + 10 (ALUMux) + 10 (RegMux) + 10 (BranchMux) + 100 (ALU) + 2000 (DM) + 500 (Control)= **3890**

The new cost would then be:

3890 + 600 = **4490**

There is a $\frac{3890-4490}{3890} = 15.42\%$ increase in cost.

Given that there is a higher price and a worse performance, we can infer that the cost/performance ratio would likely be worse.

$$Cost/Performance = \frac{cost}{performance} = \frac{4490 * (1630 * 0.95I)}{3890 * (1330 * I)} = 1.39 \tag{2}$$

We do indeed see that the cost/performance ratio of the proposed implementation is 1.39 times higher than the original implementation.

Question 4.4.1

If we only ever needed to fetch instructions, the cycle time would be **200ps**. We do not need to account for the adder because the processes occur in parallel.

Question 4.4.2

To carry out an unconditional, PC-relative branch, the cycle time would be:

200ps (IM) + 15ps (Sign Extend) + 10ps (Shift Left) + 70ps (Add) + 20ps (MUX) = **315ps**

Question 4.4.3

To carry out a conditional, PC-relative branch, the cycle time would be:

200ps (IM) + 90ps (Regs) + 20ps (ALUMUX) + 90ps (ALU) + 20ps (PCMUX) = **420ps**

We replace the sign-extend path (15ps) with the register file path (90ps). We also replace the shift left and add path (70ps+10ps) with the ALU time (90ps). Finally, we add an extra MUX (20ps) before the ALU.

Question 4.4.4

The shift-left-2 (sl2) element simulates a multiplication by four. Hence, PC-related branch instructions, which multiply the immediate value by 4 to produce an offset, make use of the sl2 element. Although, Figure 4.11 does not support jump instructions, they also multiply the 26-bit value by 4 using the sl2 element to obtain 28 bits, which are bottom-concatenated with the first four bits of the PC. It should also be noted that jump instructions don't use that specific sl2 element shown in Figure 4.11.

Question 4.4.5

The sl2 element is not on the critical path for any instructions. In branch instructions, the critical path involves obtaining the MUX selector from the ALUZero output. In jump instructions, the final value "waits" on the PC+4 adder (see Figure 4.24).